

Presentationalconponents

Presentation Overview ■ Presentational components are the atomic components in the PlayPlus architecture that focus purely on rendering and user interaction. They receive data through inputs and emit events through outputs, making them highly reusable and testable.

Key Characteristics ■ **Pure Components** ■ Focus solely on presentation and user interaction No business logic or data fetching Receive all data through inputs Emit events through outputs

Unidirectional Data Flow ■ Data flows in through inputs Events flow out through outputs No direct service dependencies Predictable and testable behavior Accessibility First ■ Built-in ARIA support Keyboard navigation Screen reader friendly Semantic HTML structure

Template Structure ■ TypeScript Component (component.ts.hbs) ■

```
import { Component , ChangeDetectionStrategy ,
input , output , computed , } from "@angular/core" ; import { CommonModule } from
"@angular/common" ; export interface ExampleData { readonly id : string ; readonly title : string ;
readonly description ? : string ; readonly isActive : boolean ; } @ Component ( { selector :
"app-example" , standalone : true , imports : [ CommonModule ] , templateUrl :
"./example.component.html" , styleUrls : [ "./example.component.scss" ] , changeDetection :
ChangeDetectionStrategy . OnPush , } ) export class ExampleComponent { // Signal inputs
(Angular 17+) readonly data = input . required < ExampleData > ( ) ; readonly customStyles =
input < Record < string , string >> ( { } ) ; readonly disabled = input < boolean > ( false ) ; // Signal
outputs (Angular 17+) readonly actionTriggered = output < string > ( ) ; readonly itemClicked =
output < ExampleData > ( ) ; // Computed values readonly isInteractive = computed ( ( ) => ! this .
disabled ( ) && this . data ( ) . isActive ) ; readonly safeStyles = computed ( ( ) => this .
sanitizeStyles ( this . customStyles ( ) ) ) ; readonly ariaLabel = computed ( ( ) => ` ${ this . data ( )
. title } - ${ this . data ( ) . description || "No description" } ` ) ; protected onAction ( ) : void { if ( this .
isInteractive ( ) ) { this . actionTriggered . emit ( this . data ( ) . id ) ; } } protected onClick ( ) : void {
if ( this . isInteractive ( ) ) { this . itemClicked . emit ( this . data ( ) ) ; } } protected onKeyDown (
event : KeyboardEvent ) : void { if ( event . key === "Enter" || event . key === " " ) { event .
preventDefault ( ) ; this . onClick ( ) ; } } private sanitizeStyles ( styles : Record < string , string > ) :
Record < string , string > { // Implement style validation logic const allowedProperties = [ "color" ,
"background-color" , "border" , "padding" , "margin" , ] ; return Object . entries ( styles ) . reduce ( (
acc , [ key , value ] ) => { if ( allowedProperties . includes ( key ) ) { acc [ key ] = value ; } return acc
; } , { } ) as Record < string , string > ; } } HTML Template ( component.html.hbs ) ■ < article class
= " example-component " [class.disabled] = " disabled() " [class.interactive] = " isInteractive() "
[attr.aria-label] = " ariaLabel() " [attr.aria-disabled] = " disabled() " role = " article " tabindex = " 0 "
(click) = " onClick() " (keydown) = " onKeyDown($event) " [style] = " safeStyles() | json " > <
header class = " component-header " > < h3 class = " component-title " > {{ data.title }} </ h3 > @if
(data.description) { < p class = " component-description " > {{ data.description }} </ p > } </ header
> < div class = " component-content " > <!-- Component content goes here --> < div class = "
```

```

content-placeholder " > < p > Component content goes here </ p > </ div > </ div > < footer class =
" component-footer " > @if (isInteractive()) { < button type = " button " class = " action-button "
(click) = " onAction(); $event.stopPropagation() " (keydown.enter) = " onAction();
$event.stopPropagation() " (keydown.space) = " onAction(); $event.stopPropagation() " aria-label
= " Trigger action for {{ data.title }} " > Action </ button > } </ footer > </ article > Features ■ Built-in
Features ■ Signal Inputs/Outputs Modern Angular 17+ signal-based inputs Type-safe data flow
Reactive updates Accessibility ARIA labels and roles Keyboard navigation support Screen reader
friendly Semantic HTML structure Style Customization Custom styles input Style sanitization for
security CSS custom properties support Interaction States Disabled state handling Interactive
state computation Event handling with proper validation Best Practices ■ Input Validation
readonly data = input . required < ExampleData > ( ) ; readonly disabled = input < boolean > (
false ) ; Computed Values readonly isInteractive = computed ( ( ) => ! this . disabled ( ) && this .
data ( ) . isActive ) ; Event Handling protected onClick ( ) : void { if ( this . isInteractive ( ) ) { this .
itemClicked . emit ( this . data ( ) ) ; } } Accessibility readonly ariaLabel = computed ( ( ) => ` ${ this
. data ( ) . title } - ${ this . data ( ) . description || 'No description' } ` ) ; Usage Examples ■ Basic
Presentational Component ■ # Generate a presentational component playplus generate
presentational user-card This creates: UserCardComponent with data input Action and click
outputs Accessibility features Storybook stories Advanced Presentational Component ■ #
Generate with custom interface playplus generate presentational product-tile --interface=Product
This creates: ProductTileComponent with Product interface Custom styling support
Comprehensive interaction handling Integration with Container Components ■ Presentational
components are used by container components: // In container component template < app - user -
card [ data ] = "userData()" [ disabled ] = "loading()" ( actionTriggered ) = "onUserAction($event)" (
itemClicked ) = "onUserSelect($event)" > < / app - user - card > // In container component class
export class UserListContainerComponent { protected users = signal < User [ ] > ( [ ] ) ; protected
userData = computed ( ( ) => this . users ( ) . map ( ( user ) => ( { id : user . id , title : user . name ,
description : user . email , isActive : user . isActive , } ) ) ) ; protected onUserAction ( userId : string
) : void { // Handle user action } protected onUserSelect ( user : User ) : void { // Handle user
selection } } Data Interface Pattern ■ Each presentational component defines its own data
interface: export interface UserCardData { readonly id : string ; readonly title : string ; readonly
description ? : string ; readonly isActive : boolean ; readonly avatar ? : string ; readonly role ? :
string ; } This ensures: Type safety Clear contract between components Easy refactoring Better
IDE support Styling and Theming ■ CSS Custom Properties ■ .example-component {
--card-background: #ffffff; --border-color: #e0e0e0; --text-color: #333333; --interactive-color:
#007bff; background: var(--card-background); border: 1px solid var(--border-color); color:
var(--text-color); } .example-component.interactive:hover { border-color: var(--interactive-color); }
Custom Styles Input ■ // In container component < app - user - card [ data ] = "userData()" [
customStyles ] = " { '--card-background' : '#f8f9fa' , '--border-color' : '#007bff' } " > < / app - user -
card > Testing ■ Presentational components include comprehensive tests: describe (
"UserCardComponent" , ( ) => { let component : UserCardComponent ; beforeEach ( ( ) => {

```

```
component = new UserCardComponent ( ) ; } ) ; it ( "should emit action when clicked" , ( ) => {  
const spy = jasmine . createSpy ( ) ; component . actionTriggered . subscribe ( spy ) ; component .  
onAction ( ) ; expect ( spy ) . toHaveBeenCalledWith ( "test-id" ) ; } ) ; it ( "should be interactive  
when not disabled and active" , ( ) => { component . data . set ( { id : "1" , title : "Test" , isActive :  
true , } ) ; component . disabled . set ( false ) ; expect ( component . isInteractive ( ) ) . toBe ( true )  
; } ) ; } ) ;
```

Architecture Benefits ■ **Reusability** : Can be used across different containers **Testability** :
Pure functions are easy to test **Maintainability** : Clear separation of concerns **Performance** :
OnPush change detection **Accessibility** : Built-in a11y features

Common Patterns ■ **List Items** ■ // User list item export interface
`UserListItemData { readonly id : string ; readonly title : string ;
readonly subtitle ? : string ; readonly avatar ? : string ; readonly isActive : boolean ; readonly
isSelected ? : boolean ; }`

Cards ■ // Product card export interface
`ProductCardData { readonly id : string ; readonly title : string ; readonly description ? : string ;
readonly price : number ; readonly image ? : string ; readonly isAvailable : boolean ; readonly rating ? : number ; }`

Forms ■ // Form field export interface
`FormFieldData { readonly id : string ; readonly label : string ; readonly placeholder ? : string ;
readonly value : string ; readonly type : "text" | "email" | "password" ;
readonly required : boolean ; readonly error ? : string ; }`

Next Steps ■ After creating a presentational component: Define the data interface for your specific use case
Customize the template with your content Add styling to match your design system Create stories for visual testing
Write tests for all interaction scenarios Use in container components for data integration

Developer Checklist ■ **Before Creating Presentational Components:** ■ Is the component purely presentational (no business logic)?
Are all inputs typed with TypeScript interfaces? Are all user interactions handled through outputs? Is OnPush change detection enabled?
Are computed values used for derived state? Is disabled state properly handled? Are all interactive elements keyboard accessible?
Do I have unit tests for all interaction scenarios? Is the component reusable across contexts? Are style customization options provided?